Faster algorithms for minimum path cover by graph decomposition

Topi Paavilainen

Master's thesis
UNIVERSITY OF HELSINKI
Department of Computer Science

Helsinki, October 26, 2017

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Laitos — Institution — Department
Faculty of Science	Department of Computer Science

Tekijä — Författare — Author
Topi Paavilainen

Työn nimi — Arbetets titel — Title

Faster algorithms for minimum path cover by graph decomposition

Oppiaine — Läroämne — Subject
Computer Science

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
Master's thesis	October 26, 2017	30

Tiivistelmä — Referat — Abstract

Minimum-cost minimum path cover is a graph-theoretic problem with an application in gene sequencing problems in bioinformatics. This thesis studies decomposing graphs as a preprocessing step for solving the minimum-cost minimum path cover problem. By decomposing graphs, we mean splitting graphs into smaller pieces. When the graph is split along the maximum anti-chains of the graph, the solution for the minimum-cost minimum path cover problem can be computed independently in the small pieces. In the end all the partial solutions are joined together to form the solution for the original graph. As a part of our decomposition pipeline, we will introduce a novel way to solve the unweighted minimum path cover problem and with that algorithm, we will also obtain a new time/space tradeoff for reachability queries in directed acyclic graphs. This thesis also includes an experimental section, where an example implementation of the decomposition is tested on randomly generated graphs. On the test graphs we do not really get a speedup with the decomposition compared to solving the same instances without the decomposition. However, from the experiments we get some insight on the parameters that affect the decomposition's performance and how the implementation could be improved.

ACM Computing Classification System (CCS):
Design and analysis of algorithms - Graph algorithms analysis
Applied computing - Life and medical sciences - Bioinformatics

Avainsanat — Nyckelord — Keywords
graph algorithms, minimum path cover, bioinformatics

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — Övriga uppgifter — Additional information

Contents

1 Introduction

This thesis focuses on decomposing graphs as a preprocessing step to make graph algorithms run faster. By *decomposing* a graph we mean splitting it into smaller pieces. Using a decomposition can give a speedup for solving algorithms, if we can solve the problem in the decomposed parts instead of the full graph. When using algorithms that take more than linear time to complete, it is theoretically faster to solve the problem independently in the small pieces and then combine the results. In practice, to achieve a speedup, the extra work caused by decomposing has to be smaller than the total time required to solve the problem.

The problem we will address in this thesis is minimum-cost minimum path cover. In this work we are restricted on directed acyclic graphs, in which this problem is solvable in polynomial time. More specifically, we are concentrating on solving this problem in graphs of small width, that is, graphs which have minimum path covers made up of few very long paths. This type of target graphs comes from multi-assembly problems in bioinformatics, where minimum-cost minimum path cover has some applications. We will use a decomposition which is based on maximum anti-chains in the graphs. Maximum anti-chains in a graph have important connections to minimum path cover, which make solving the problem in decomposed pieces equivalent to solving the problem in the original graph.

To measure the speedup given by the decomposition in practice, I have programmed an example implementation of the decomposition method presented in this thesis. The code is open source and freely available on Github (https://github.com/tobtobtob/MC-MPC). The algorithm itself is written in C++ that can be compiled and used as a command-line tool. The repository includes also few other tools for testing, such as random test graph generator and splitter tool for splitting a graph into connected components. The repository includes example Python scripts that use this decomposition. Those scripts are used to evaluate the performance of the decomposition and to give an example how the decomposition tools can be used.

1.1 Multi-assembly problems

The motivation to study minimum path cover in directed acyclic graphs comes from gene assembly problems in bioinformatics. The current DNA

sequencing technology cannot read the whole DNA string as whole, but instead reads small fragments of the full DNA. These fragments have to be merged together in order to reconstruct the original DNA sequence. DNA sequencing can also be applied to sequence the several RNA transcripts produced by a gene. One application of the minimum-cost minimum path cover problem to assembling RNA transcripts is as follows. After sequencing the RNA transcripts, one aligns the RNA fragments to a reference genome of that species. One then models the RNA fragments as nodes of a graph, and any overlap between their alignments as an edge of the graph. Since the alignments are to a reference, then the resulting graph is acyclic. Finally, one asks for the minimum number of paths that cover all the RNA fragments and has minimum overall cost (where some costs are also attached to the edges). Since we are interested in this problem from an algorithmic perspective, we refer the reader to [12] for further bioinformatics details.

1.2 Contributions

The idea of decomposing directed acyclic graphs along its maximum antichains was from my thesis supervisor Alexandru Tomescu. Together with Ruxandra Barbulescu they started a project to find out if the decomposition could be used to speed up minimum path cover solvers. They implemented the first version of the decomposer, but the decomposition was not fast enough to give performance boost to solvers. To be usable, the decomposition step has to be faster than the solver, otherwise there is no point using the decomposition. My goal was to explore the optimization possibilities for this decomposition, so that decomposition could be used to solve the problem faster.

We will present a method to decompose directed acyclic graphs in a way that minimum-cost minimum path cover problem can be solved in the decomposed parts instead of the full graph. To create the decomposition, we need to solve the unweighted minimum path cover problem. We will present a new way to solve the minimum path cover problem in $O(k|E|\log|V|)$ time, where k is the size of the minimum path cover. Using this algorithm, we will present a new space/time tradeoff for reachability queries in directed acyclic graphs. The new minimum path cover algorithm allows us to answer reachability queries on directed acyclic graphs in $O(1)$ time, after constructing an index of $O(k|V|)$ size in $O(k|E|\log|V|)$ time.

1.3 Structure

We will start by going through all the graph-theoretic concepts that are used in the decomposition. In Section 3 we will go through the decomposition method in detail, explaining all the steps as they are implemented in the example implementation. The following section will contain the results of the tests conducted on the example implementation. We will compare solving the same problem with and without the decomposition and see when the decomposition really gives us a speedup. The tests are conducted on different kinds of generated graphs, that resemble the graphs in our application domain. With generated graphs we can adjust the parameters easily and see what parameters of graphs affect the usability of the decomposition. Even though graphs in the application domain would not have those parameters, these experiments can show us potential other application areas for this decomposition. The last section includes a discussion and thoughts about the results, as well as possible improvement ideas.

2 Preliminaries

In this section we will go through the basic graph-theoretic concepts, problems and results which are relevant for this thesis. We will start by defining graphs and the notation used in this thesis and go through path covers, flows and anti-chain decompositions for directed acyclic graphs. We will also introduce some set theoretic concepts, because we will use some insight from the set covering problem in our decomposition. We will use flow algorithms after the decomposition to solve the problem. Flows and path covers are not exactly equal problems, but have many similarities and for that reason, with small modifications algorithms for minimum flow can be used to solve minimum path cover. Anti-chains are the main component of the decomposition. We will also introduce partially ordered sets, which are a more general definition of directed acyclic graphs.

2.1 Graphs, nodes and arcs

A graph $G = (V, E)$ consists of a set of nodes, V, and of a set of edges between the nodes, E. An *edge* is a pair of nodes (n_1, n_2), where n_1 and n_2 are called the *endpoints* of the edge. An edge can be either directed or

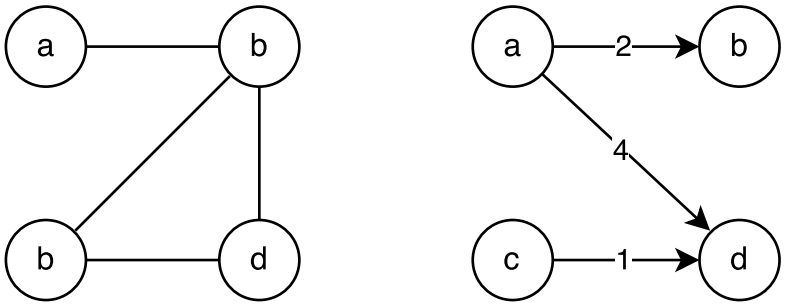

Figure 1: An undirected graph (left) and a directed graph (right) with arc weights

undirected. The direction of an edge is denoted by an arrowhead and is interpreted so that the edge can be traversed only in that direction. To differentiate between directed and undirected edges, edges with direction are called arcs. Graphs in this thesis are always directed, so from now on we will only use the word arc. Arcs can have *costs* or *weights*, which is a numerical value attached to each arc. In many graph problems we want to minimize the sum of costs of arcs used in the solution. A *path* in a directed graph is a sequence of nodes $(n_1, n_2, n_3, ..., n_n)$ so that there is an arc from every n_i to n_{i+1}. Each node can appear only once in a path. Figure 1 shows examples of undirected and directed graphs.

In this thesis we consider only directed acyclic graphs (DAG). A graph is directed if every edge has direction, and acyclic if there are no cycles. A directed graph has a cycle if there are nodes n_1 and n_2 so that there is a path from n_1 to n_2 and from n_2 to n_1. In particular, in acyclic graphs no node can be reached from itself.

2.2 Path cover

The goal of this thesis is to find a more efficient solver for the minimum-cost minimum path cover problem. A *path cover* is a set of paths so that every node in the graph belongs to at least one of the paths. A *minimum path cover* is a path cover with minimum number of paths, and a *minimum-cost minimum path cover* is a minimum path cover where the sum of weights of paths is minimized. The *weight* of a path is equal to the sum of weight of the arcs included in the path. The minimum-cost minimum path cover is a

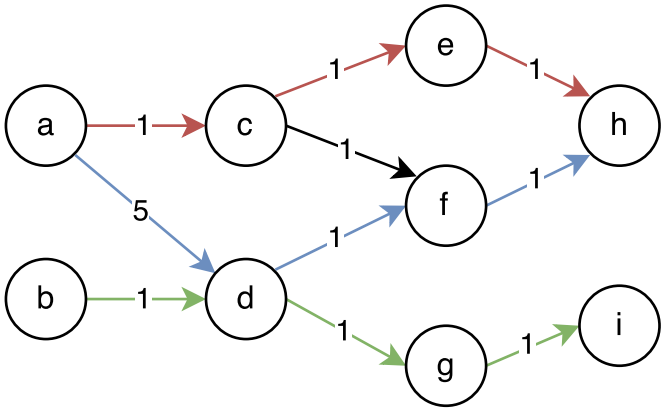

Figure 2: A minimum source-to-sink path cover (colored arcs)

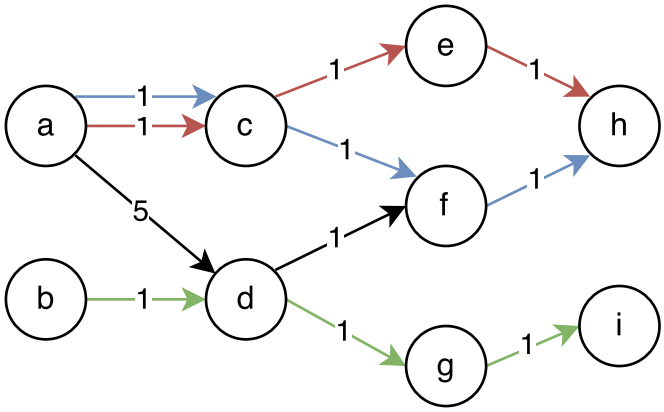

Figure 3: A minimum-cost minimum source-to-sink path cover (colored arcs)

strictly more general problem than minimum path cover; every minimum-cost minimum path cover is also a minimum path cover, but not vice versa. With minimum path cover we can define *width* for directed acyclic graphs. The width of a DAG is the size of the minimum path cover of the graph.

In the general case of graphs with cycles minimum path cover is an NP-hard[1] problem. Hamiltonian Path[2], which is a commonly known NP-hard problem, can be reduced to a minimum path cover instance. There is a Hamiltonian Path in a graph if and only if the size of the minimum path

[1]NP-hard, where NP stands for non-deterministic polynomial, is a class of problems, which are believed to be not solvable in polynomial time.

[2]The Hamiltonian path problem asks if there is a path which covers all the nodes in a graph.

cover for the graph is one. However, in directed acyclic graphs, the problem is solvable in polynomial time. We will see in Section 2.5 how a path cover can be solved as a minimum flow [15], which is solvable in polynomial time. Minimum flows can be solved as maximum flows, and maximum flows can be solved in $O(|V||E|)$ time [14]. Another method to solve minimum path cover is to reduce it to a maximum matching problem [8]. Maximum matchings can be solved in $O(t(G)+\sqrt{|V|T(G)})$ time by using the Hopcroft-Karp algorithm [10], where $t(G)$ is the time needed to compute the transitive closure of the graph, and $T(G)$ is the size of the transitive closure.

Both of these algorithms for solving minimum path cover have running times independent of the width of the graph. Our algorithm for minimum path cover runs in $O(k|E|\log V)$ time, where k is the width of the graph. This is worse in the class of all DAGs, because the width can be as big as the number of nodes, but becomes better in our application area, where widths of the graphs are small. We are aware of two previous algorithms for minimum path cover parametrized by the width, both by Chen and Chen. The first one runs in $O(|V|^2 + k\sqrt{k}|V|)$ time [2] and the second one in $O(max(\sqrt{|V||E|}, k\sqrt{k}|V|))$ time [3]. Depending on the value of k, our algorithm can run better than the previous algorithms by Chen and Chen. Our algorithm is explained in Section 3, where we go through the whole decomposition pipeline in detail.

From now on, we will talk only about *source-to-sink path covers*. A *source* is a node with no incoming arcs and a *sink* is correspondingly a node with no outgoing arcs. A *source-to-sink path cover* is a path cover where all the paths start in sources and end in sinks. Source-to-sink path cover is a more general problem than the minimum path cover problem, both in unweighted and weighted cases. Indeed, we can reduce an instance of path cover problem to source-to-sink path cover by adding a global source and sink nodes and adding an arc from the global source to every node in a graph and an arc from every node to the global sink. In this way the paths in source-to-sink path cover (excluding the global sink and source added) in this new graph correspond to a minimum path cover in the original graph.

2.3 Set Cover

Path cover is a covering problem for graphs; we are trying to find a collection of paths that cover all the nodes in the graph. Covering problems appear

also in other fields of mathematics. Set cover is a covering problem where we cover the elements of a universe with some given sets. We can show that path cover is a special case of set cover, and algorithms for set cover can be applied to path cover problems. In set cover we define a universe of elements A, and a set of subsets $S = \{S_1, S_2, S_3, ..., S_m\}$ where every S_i is a subset of A. A solution to set cover is a selection of subsets in S, so that the union of subsets is equal to A. A trivial solution, that covers all the elements in the universe is to include all the subsets. The goal is to find a set cover of smallest size. In the weighted case, every subset is assigned a numerical weight and we are trying to find a set cover of smallest total weight.

Set cover is also an NP-hard problem. However, there is a $\ln n$-approximation algorithm, which can be found in textbooks such as [17]. This algorithm is guaranteed to create a set cover at most $\ln n$ times the size of the optimal solution by choosing subsets greedily. The algorithm is executed in rounds, adding one subset to the set cover every round. Each round we select the subset which covers most previously uncovered elements. This is continued until there are no more uncovered elements. The following theorem is a classical result, appearing for example in [17]. We include the proof here for the sake of completeness.

Theorem 1. *The Greedy algorithm for the unweighted set cover problem has approximation factor of* $\ln n$*, where* n *is the number of elements in the universe.*

Proof. Assume we have a universe of n elements with an optimal set cover of size k. Every round we pick the subset which covers most uncovered elements, until the chosen subsets cover the whole universe. Each new subset will cover at least $\frac{u}{k}$ new elements, where u is the number of uncovered elements left. This follows from the fact that the optimal solution can cover the universe with k sets, so at least one of the sets has to cover at least $\frac{n}{k}$ elements, otherwise a set cover of size k would not exist.

To prove that the algorithm achieves the approximation factor of $\ln n$ we define a cost for each element. Let the cost of each element be $\frac{1}{C}$, where C is the number of previously uncovered elements in the subset which added the element the first time. This way the total cost for each subset is 1 and the total cost for a solution is equal to the amount of subsets included in the set cover. The optimal solution is of size k, so the total cost of the optimal

solution will also be k. Now we will show that the cost of any solution found with this algorithm will always be at most $k \ln n$.

Let us now consider adding new elements individually. An element is considered to be added when it is covered the first time. Even though we add elements in groups, for the purpose of the analysis, we will consider adding each element separately. We will use the cost for elements defined above. When adding each element, let us say that we have u uncovered elements left. Then the element's cost can be at most $\frac{k}{n-u+1}$. This follows from the fact that the optimal solution can always cover the remaining elements with cost k (it can cover all elements with cost k, so any subset of the elements can also be covered with that cost), so at least one of the subsets has to have this cost, otherwise not even the optimal solution could cover the remaining elements with cost k. When we sum the costs of all elements, we get

$$\sum_{u=0}^{n-1} \frac{k}{n-u+1}$$

$$= k \sum_{u=0}^{n-1} \frac{1}{n-u+1}$$

$$= k \sum_{i=1}^{n} \frac{1}{i} \approx k \ln n.$$

The sum of the costs is equal to sum of a harmonic series, which is known to be an approximation of natural logarithm. Because of the way how we defined the cost for elements, the cost is equal to the size of the set cover, so the approximation of the set cover is always at most $\ln n$ times the size of the optimal solution.

□

As mentioned above, path cover is a special case of set cover, and for that reason we can use the same greedy algorithm for set cover to get an $\ln n$ approximation of the minimum path cover. Let us first formulate path cover as an instance of set cover. The universe U is the set of all nodes. The set S of subsets is a set of all possible source-to-sink paths. Due to the ordering of nodes in directed acyclic graphs, a set of nodes can only form one path (if any), so we can represent paths as sets of nodes.

Then we execute the algorithm in rounds, always choosing the path

with most uncovered nodes. We can find the path with most uncovered nodes efficiently by maintaining a data structure which tells us the number of uncovered nodes in the path with most uncovered nodes starting from each node. This data structure will be explained in detail in later chapters. Because path cover fulfills all the definitions of set cover, we can say without further proofs that this greedy algorithm achieves the same approximation factor of $\ln n$.

2.4 Flow Problems

Flow problems are a class of problems in directed graphs. The idea is to push flow from a source node to a sink node along the arcs of the graph. A common analogy is that arcs in the graph are pipes and we try to find a flow of water from source to sink using the pipes. A solution to a flow problem is a numerical value for each arc that tells the amount of flow going through that arc. Any solution has to satisfy *flow conservation*, which means that the amount of flow coming to a node through inbound arcs has to be equal to the amount of flow leaving the node through outbound arcs. An exception to flow conservation rule is that sink and source nodes do not have to satisfy the condition. We can define flow problems as source-to-sink flows as above or as circulation problems. In the circulation setting, there are neither source nor sink nodes; instead, flow conservation has to apply in each node. The circulation in acyclic graphs is always zero, so to use circulation we must have a graph with at least one cycle. Arcs in flow network may or must have (depending on the problem type) capacities, demands and weights, which are numerical values attached to each arc. The cost of a flow or a circulation is the sum, over all arcs of the graph, of the flow value of the arc multiplied by its weight.

Maximum flow is a common flow maximization problem. Each arc is assigned a capacity value, which is the maximum amount of flow that can pass through that arc. The objective is to find the maximum amount of flow possible from source to sink, so that no arc carries more flow than its capacity allows. In Figure 4 we have a directed acyclic graph with capacities in red and a maximum flow in blue. This graph has a maximum flow of four. Maximum flow can be solved in polynomial time. Many other graph problems, like maximum cut and maximum matching in bipartite graphs can be solved as a maximum flow instance. Later we will use maximum flow

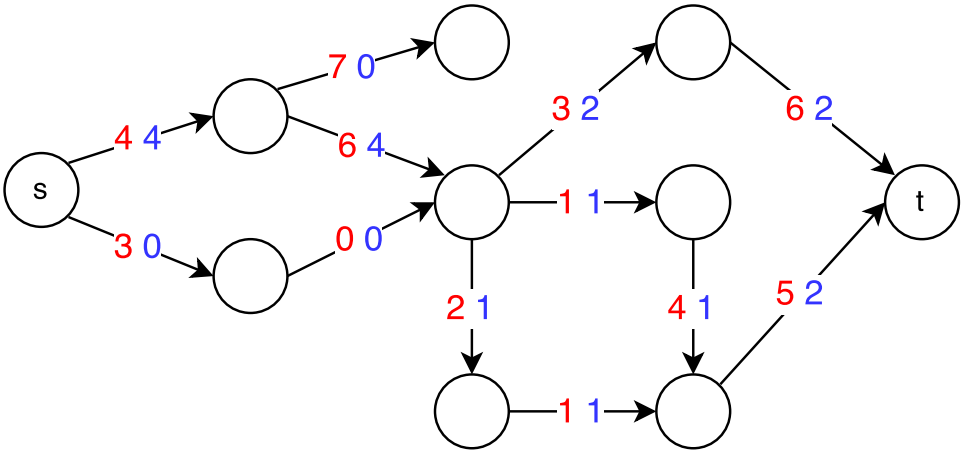

Figure 4: A directed acyclic graph with arc capacities (red numbers) and a maximum flow from s to t (blue numbers)

to solve the unweighted path cover problem.

Minimum flow is a related problem, but this time the object is to find the minimum possible flow. Minimum flow with just arc capacities would be just a flow with zero amount, so we have demands on arcs, instead of capacities. Demand is a numerical value, and for a flow to be feasible, the flow value on each arc has to be equal or greater than its demand. There is a minimum flow pictured in Figure 5 with demands but no capacities on arcs. Minimum flow can also be solved in polynomial time.

2.5 Solving minimum-cost path cover as minimum circulation

In this thesis we will solve minimum-cost minimum path cover as a minimum flow problem, using the reduction given in [12]. We will describe this reduction also here for completeness. Each unit of flow on each arc corresponds to one path going through that node. Due to flow conservation, all paths will be source-to-sink paths as desired, as no paths/flow can be lost on nodes. To solve the path cover as a minimum flow, we will define a construction, which transforms a directed acyclic graph to a cyclic directed graph, where minimum flow circulation can be solved. This construction is pictured in Figure 6 and explained below.

To make sure that the path cover is a feasible cover and covers all nodes, we split each node v in two nodes, v_1 and v_2. We attach each inbound arc to

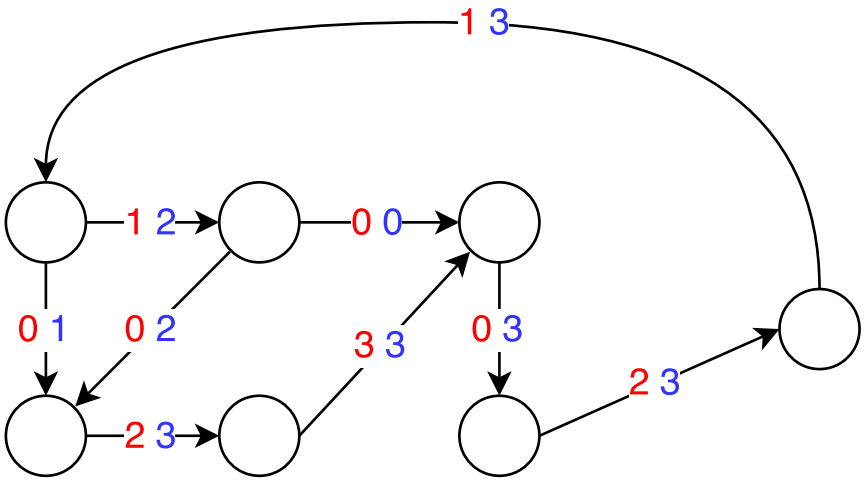

Figure 5: A directed graph with arc demands (red) and a minimum circulation flow (blue)

v_1 and each outbound arc to v_2. Finally we add an arc with demand one from v_1 to v_2 to make sure that at least one path goes through the node. We also add a global source and sink nodes, and connect these nodes to the sink and source nodes of the original graph, respectively.

Until this point, the graph has been acyclic. To use minimum flow circulation we add an arc from the global sink to the global source. We set the weight of this arc to be equal to the sum of all other arcs' weights plus one. The purpose of this arc is to force the minimum flow use the minimum number of paths. Due to the weight of this sink-to-source arc, any path in the original graph has lower cost than the sink-to-source arc, so the number of paths will be minimized.

2.6 Partially ordered sets, chains and anti-chains

A *partially ordered set*, or a *poset*, consists of a set of elements (also called the ground set) and a partial ordering relation, denoted $<$, between the elements. Partiality of the ordering means that not all the two element pairs of the set have to be comparable. Partial ordering has to be reflexive, symmetric and transitive. An element a in a partially ordered set S is called *maximal*, if for all elements comparable with a $x \in S$ $x < a$. A *chain* in a partially ordered set is a subset of elements which all are comparable with each other.

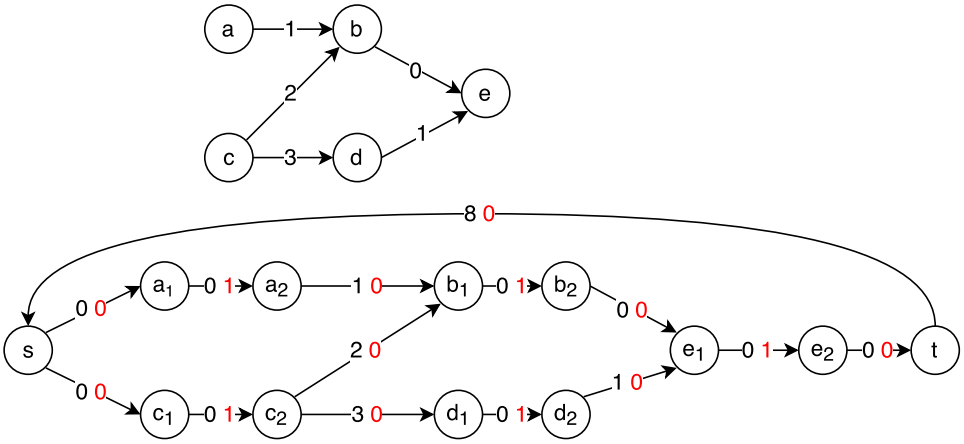

Figure 6: A directed acyclic graph and a construction to solve minimum-cost minimum path cover in it as a minimum flow circulation. In the construction, black numbers are arc weights and red numbers arc demands.

Inside a chain there is a complete ordering of elements. An *anti-chain* is the opposite of a chain. All elements in an anti-chain are mutually incomparable. An anti-chain is *maximal*, if it is not a proper subset of any other anti-chain, and *maximum*, if there are no bigger maximal anti-chains. Note that in a partially ordered set there can be more than one maximum anti-chain; all the anti-chains with maximum size are maximum anti-chains.

As we did with set cover previously, we will again extend a set theoretic concept to graph theory. Directed acyclic graphs are practically partially ordered sets. We can use reachability between two nodes as an ordering. For any two nodes v_1 and v_2, we define $v_1 < v_2$ if and only if v_2 is reachable from v_1. Acyclicity guarantees that only one node can be reachable from the other node. If neither of the nodes is reachable from the other node, then the nodes are incomparable. A chain corresponds to a path and anti-chain in a directed acyclic graph is a set of nodes that all are mutually unreachable.

The next theorem below shows a connection between anti-chains and minimum path cover. When talking about partially ordered sets in general, we use the term chain-cover, which is a set of chains covering the set, exactly like a path cover. We will use this connection to find a decomposition which makes minimum path cover easier to solve. The proof is due to Galvin [9] and is included here for the sake of completeness.

Theorem 2 (Dilworth's theorem [5])**.** *The size M of a maximum anti-chain*

in a partially ordered set S is equal to the size m of the minimum set of chains covering the set.

Proof. To show that $m = M$ in a partially ordered set, we will first show that $M \leq m$. Each chain in a set of chains covering S can only cover one element of an anti-chain. Thus the size of a chain cover has to be at least equal to the number of elements in the maximum anti-chain.

We will now prove $m \leq M$; that a partially ordered set S with a maximum anti-chain of size M can be covered by M chains. We will prove this direction with induction on the size of the set S. The base case for $|S| = 1$ is trivially true, because chain cover and maximum anti-chain are equal. We will show that for a partially ordered set P, $m \leq M$ assuming that the induction hypothesis holds for all sizes smaller than $|P|$.

If all elements in P are incomparable, the case is trivially true, because every element belongs to the one anti-chain and every element is also a chain of its own. Consider the case where not all elements are incomparable. We pick an element a so that a is a maximal element in P. Consider the poset $P' = P \setminus \{a\}$ with a maximum anti-chain of size k. By the induction hypothesis P' can be split into k chains $C_1, C_2, ..., C_k$. Every maximum anti-chain in P' consists of one element of each of the chains (otherwise it would not be an anti-chain).

Consider a set $K = \{k_1, k_2, ..., k_k\}$. Each element in K is the maximal element of each chain $C_1, C_2, ..., C_k$, which also belong to some maximum anti-chain in P'. It can be proven that K is also a maximum anti-chain. By contradiction, assume that K is not an anti-chain and $k_i \leq k_j$ for some elements in K. By definition of K, k_i belongs to some maximum anti-chain A_i and k_j belongs to some maximum anti-chain A_j. Because A_j is a maximum anti-chain, A_j must intersect the chain C_i, and there is an element x which belongs to both A_j and C_i. We defined k_i to be the maximal element belonging to an anti-chain in C_i, so $x \leq k_i$. By transitivity, also $x \leq k_j$. But this is a contradiction, because x and k_j were supposed to be in the same anti-chain A_j.

Now consider the original poset P. If $K \cup \{a\}$ is an anti-chain in P, then $M = k + 1$, and P can be covered with chains $C_1, C_2, ..., C_k$ and $\{a\}$. If not, then $k_i < a$ for some i. Then we have a chain $C' = \{c \in C_i |\ c \leq a\} \cup \{a\}$. Because k_i was a maximal element of C_i belonging to some maximum anti-chain, poset $P \setminus C'$ has a maximum anti-chain of size $M - 1$ and by induction

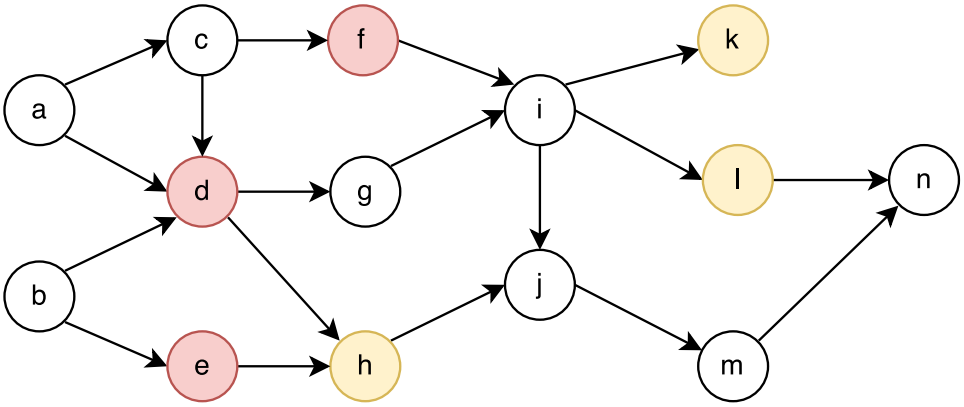

Figure 7: A directed acyclic graph with two maximum anti-chains

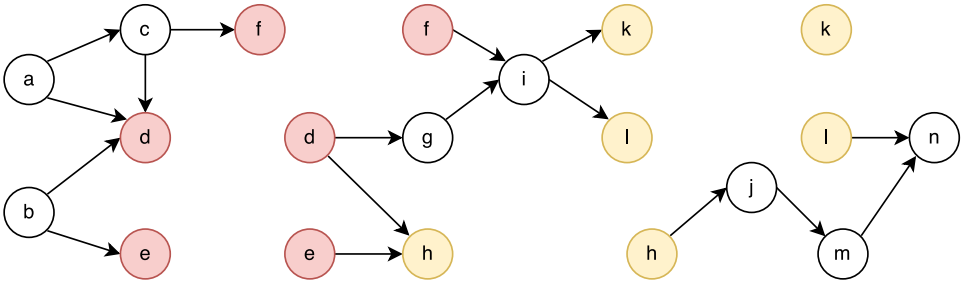

Figure 8: The graph from Figure 7 decomposed along its maximum anti-chains. Note that the arc from i to j spans over two sections and is discarded in the decomposition.

assumption can be covered be $M - 1$ chains. Then together with $M - 1$ chains covering $P \setminus C'$ and C', the poset P can be covered with M chains.

□

2.7 Decomposition along maximum anti-chains

Maximum anti-chains serve us as points where to cut the graph in pieces. In Figure 7 we have a graph with two maximum anti-chains, colored in red and yellow. We do the decomposition by splitting every node belonging to an anti-chain in two. If there are arcs that span over two decomposition parts, they are discarded. In Figure 8 we have decomposed the graph along its two maximum anti-chains. In the original graph there is arc from i to j, but in the decomposition it is discarded.

Theorem 3. *Solving minimum-cost minimum path cover in decomposed parts gives us the same result as solving it in the original graph.*

Proof. Due to Dilworth's theorem (Theorem 2), the size of a maximum anti-chain and of a minimum path cover are equal. From this we get that in a minimum path cover, there is exactly one path going through every node belonging to maximum anti-chain. If there were more than one path in one node, another node in the maximum anti-chain would be missing a path, and we would need a bigger path-cover, which contradicts the theorem. For the same reason we know that arcs spanning over two decomposition sections cannot be used in minimum path cover; such path would skip a node belonging to a maximum anti-chain, and we would again need a bigger path cover. Because we know the locations of paths in an anti-chain, we can solve the minimum-cost minimum path cover between two maximum anti-chains independently and be sure that the result is equal to when solving the same problem in the full graph.

□

3 Decomposition

In this section we will go through the decomposition process in detail. In the previous section we went through all the graph-theoretic concepts required. This part will focus on the implementation details and give a detailed look into the decomposition process. The decomposition process consists of many independent steps which are executed in order to decompose the input graph. The process is pictured in Figure 9. First we have to find a minimum path cover (unweighted) in the graph. This minimum path cover is found by creating a small feasible path cover - path cover which is not minimum, but close to that. Then this feasible path cover is minimized to a minimum path cover by finding diminishing paths, which still retain the covering requirement.

3.1 An $\ln|V|$ minimum path cover approximation

We use the classic greedy set cover approximation algorithm introduced in Section 2.3 to find an $\ln|V|$ approximation of the minimum set cover. We will create the path cover iteratively by always choosing the path which

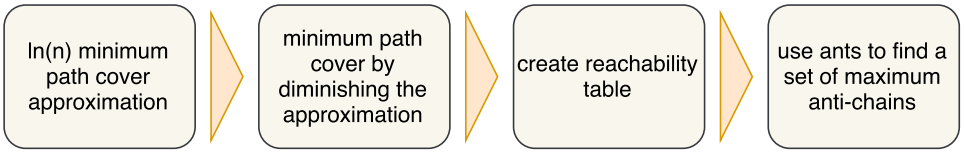

Figure 9: Decomposition process

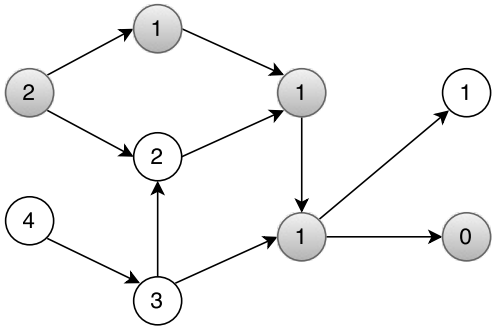

Figure 10: The state between two iterations of finding the approximation of minimum path cover. Grey nodes are already covered, white are uncovered and the values on nodes are the amount of uncovered nodes in the most-uncovered path starting from the node.

contains most uncovered nodes, until the set of paths cover all the nodes. Two issues in this approach are that the set of all paths is big, and how to calculate the amount of uncovered nodes for each path? To address these issues we will maintain a table of values A and for each $v \in V$, $A(v)$ is the maximum amount of uncovered nodes on any path starting from v. Initializing these values for A is straightforward. We will process the nodes in reverse topological order[3], starting from the sinks. Values for the sinks will be initialized to 1, as the sinks themselves are still uncovered at the start and they have no children nodes. Then for each $v \in V$, when v is processed, we set $A(v)$ to be $\max(\{A(c) \mid (v, c) \in E\}) + 1$. The 'plus one' comes from the fact that the node itself is uncovered.

After calculating the initial values for A, we will construct the approximation path cover by selecting paths one by one from the graph until all the nodes are covered. Paths are selected by traversing through the graph

[3]Topological order is an ordering of partially ordered elements, like directed acyclic graphs, where element x comes before element y if and only if $x < y$.

and always picking the next node to be the children with highest value in A. When we arrive to a sink, the path is complete and is added to the path cover. When a path is found, we have to update A, because by adding a path to the path cover new nodes are covered. We update A almost as when initializing the values, but add 'plus one' if and only if the node is uncovered. To find out quickly if a node is covered or not, we maintain a table of boolean values C, and $C(v)$ is set to true when v is selected to a path for the first time.

Theorem 4. *Using the greedy algorithm for set cover, we can find a* $\log |V|$ *approximation of the minimum path cover in* $O(k|E| \log |V|)$ *time.*

Proof. Finding each most-uncovered-nodes path takes $O(|E|)$ time. The greedy algorithm for set cover is guaranteed to create a path cover of at most $O(k \log |V|)$ paths. Amount of paths multiplied by the time required for finding one path makes the time complexity $O(k|E| \log |V|)$. □

3.2 Diminishing the approximation to a minimum path cover

The second part in obtaining the minimum flow is to diminish the approximation into a minimum flow. We will use a method that is very close to Ford-Fulkerson method [7] for finding maximum flows. In Ford-Fulkerson method the maximum flow is found by finding paths from source to sink and adding each found path to the total flow. While finding the paths, directed edges can be used in both directions. When an arc is used in its correct direction, we are adding flow on the arc. Correspondingly, if an arc is used in backward direction, we subtract flow from that arc. When using arcs in forward direction, arc capacities have to be respected. The flow on an arc cannot be reduced under zero, so to use an arc in backward direction, there has to be some flow on that arc initially. When no more paths can be found, we have found a maximum flow.

We use the same idea to diminish the approximation to a minimum path cover. Even though Ford-Fulkerson is a method for solving flows, it makes no difference when solving path covers. The important thing is flow conservation; because flow is conserved, every unit of flow on each arc corresponds to one path going through that arc. First we need a global source connected to all sources, as well as global sink which is connected from all the sinks. Then we find paths from the source to sink, until no more paths can be found. This

is done almost like in normal Ford-Fulkerson, except that we reduce flow on forward arcs and increase flow on backward arcs. When no more paths can be found, the minimization is ready.

One difference to flows is that we have to have at least one path going through each node. This could be solved by splitting each node in two and adding an arc with demand one, as we did with minimum flows in Section 2.5. However, creating new nodes to a graph causes always some overhead, so in the example implementation this is avoided. Instead, when finding paths, we are not allowed to move forward from a node if that would leave node without a path going through it. Note that we can always use backward arcs, because using backward arcs means that there is new flow coming to the node.

Theorem 5. *We can find a minimum path cover in directed acyclic graphs in $O(k|E|\log|V|)$ time with the diminishing method described above.*

Proof. Finding each path takes $O(|E|)$ time, because to find the path or impossibility of finding more paths we have to check at most all the arcs in the graph. The number of paths to find is the difference between the size of the approximation and the optimal solution. This difference is at most $k\log|V|$, which is the size of the approximation obtained with the greedy algorithm. That makes the final time complexity for the diminishing algorithm $O(k|E|\log|V|)$. □

3.3 Finding a set of maximum anti-chains

We use a greedy method based on minimum path cover obtained in previous steps to find a set of maximum anti-chains. From Dilworths' theorem we know that every maximum anti-chain has exactly one node on each path. We traverse the graph having one pointer for each of the paths. We call the pointers *ants*, as the pointers resemble a swarm of ants walking in the graph, each ant walking its own path in the path cover. We move ants so that every time an ant can reach another ant, it is moved forward along its path so long that it can no longer reach any ant. Then we pick another ant and move it in the same way. After any ant is moved, we check if the current configuration of ants is a maximum anti-chain. If it is, we save that configuration as a maximum anti-chain and move every ant one step forward. If it is not, there has to be at least one ant that can reach another ant and we continue by

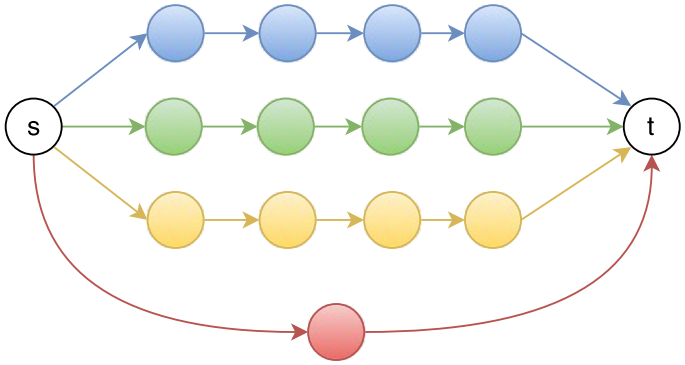

Figure 11: An example of a bad case for decomposition using ants approach. Only one maximum anti-chain will be found in this graph

moving that ant forward as described earlier. During moving the ants we have to make a lot of reachability queries. This issue is addressed in the next section.

This method is simple and efficient, but it gives no guarantees on the amount of maximum anti-chains found in the graph. Usually this is not a problem, but there are some scenarios, where this approach cannot find a good amount of maximum anti-chains, even though there would be many in the graph. There is an example of that kind of graph in Figure 11. Even though there would be many possible anti-chains, the ants approach will find only one. After finding the first maximum anti-chain, the ant on the red path will move to the global sink node and no more anti-chains can be found, because all the other ants can reach the red ant.

The performance on these type of graphs could be improved, but it would mean making more assumptions on the structure of the input graph. Another improvement could be to find the maximum anti-chains in a more intelligent way. The ants approach just chooses anti-chains greedily as it finds them, without any consideration. However, making more clever guesses on anti-chains consumes more running time. Input graphs are big and to get a benefit from decomposing the graph, the decomposition itself has to be fast.

3.4 Reachability queries

Reachability queries are a common and well-studied problem in graphs. A recent survey [16] presents many best solutions for reachability queries. A reachability query is a query on two nodes, v_1 and v_2, and we are asking whether there is a path from v_1 to v_2. Two naive approaches to this problem show the two opposites, between which we have to balance to find the most suitable solution for a specific application. The first approach is to maintain a full reachability table, where there is a boolean value for each pair of nodes, which tells if the first node is reachable from another. With this approach we can do reachability queries in $O(1)$ time, but requires us to keep a $O(|V|^2)$ table in memory, which can be too much for large graphs. Constructing the reachability table also takes $O(|V||E|)$ time. The opposite of this is to calculate the result every time a reachability query is done. This approach takes no space in memory, but every query has running time of $O(|V| + |E|)$ by doing for example a depth-first search in the graph.

As can be seen from above, in the two naive approaches to this problem we are balancing between space in time. We can either do pre-computation and do reachability queries in constant time, or move the computation work on queries. In this project we can exploit the fact that the graphs in our application area have minimum path covers of small size. As will be explained below, this allows us to reduce the size of the reachability table while still keeping query time constant.

3.5 A $k|V|$ reachability table

We will use the minimum path cover to create reachability table that uses $k|V|$ space. As mentioned earlier, the target graphs for this decomposition are ones with minimum path covers of few long paths. This makes it very favorable for us to parametrize the size by k, because we can treat k almost as a constant, or at least as a sub-linear function in $|V|$. The idea behind this reachability table is instead of marking nodes' reachability to every other node, mark the index of the first node that is reachable from this node in each path. This way queries are fast, and we have to keep only k integers in memory for each node.

This approach was proposed first in [2]. One difference is that in this work we use non-disjoint paths, while in [2] the cover is made of disjoint chains.

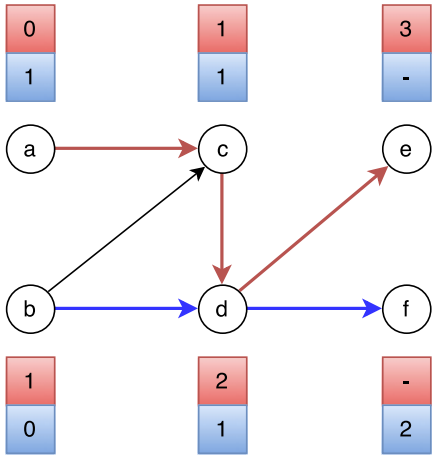

Figure 12: A directed acyclic graph with minimum path cover of size two. Reachability values for each node are marked next to each node with matching path colors. Note that nodes e and f cannot reach any nodes of the another path, so the reachability value is undefined.

The difference is that one node belongs only to one chain and chains are not necessarily paths. Disjoint chains can be seen as paths where we are allowed to skip some of the nodes. Recall Section 2.6 where we discussed chains in partial ordered sets in more detail. For the reachability table, including some nodes several times as result of using non-disjoint paths is unnecessary, but in our decomposition pipeline, we will utilize the minimum path cover also to find the maximum anti-chains in the graph. For this reason it makes sense to use the same minimum path cover for the reachability table, even if it would cause some small overhead in memory requirements.

In [2], they construct the reachability table in $O(|V|^2 + k\sqrt{k}|V|)$ time. We can use the algorithm for minimum path cover introduced in this thesis to construct the reachability table in $O(k|E| \log V)$ time. Even though this new construction time is not better in arbitrary graphs, we can argue that in sparse graphs, where $|E|$ is low, this new construction time can provide a better running time.

In Figure 12 we picture a directed acyclic graph and the reachability values for it. We calculate the reachability table by first setting the initial values. For each node, we set the initial value for each path the node is included in to be the node's own path index in that path. The first node that any node can reach on a path it belongs to is itself. All the other values

should be set to undefined or to some value larger than any possible path index. We will calculate this reachability table in a similar fashion as we calculated/updated the most-uncovered path values in the previous step, processing the graph in order from sinks to sources.

Then, for each node we can find the remaining values by finding the smallest values for this path among the child nodes of the node. When we process the nodes in reversed topological order, the values for the child nodes will have been computed when processing the actual node. We will start from the sink nodes, where all path values are final from the start and end in the sources. Time complexity for creating the reachability table consists of the topological sort $O(|V| + |E|)$ [11] plus the calculation of $O(k|E|)$ values. The number of arcs, $|E|$, is also the number of child nodes checked, and in each child node we have to check k values. As long as $k > 1$, the time complexity is dominated by $O(k|E|)$ so this is the final time complexity for creating the reachability table.

Theorem 6. *Creating a $k|V|$ reachability table with the method described above allows us to answer reachability queries in $O(1)$ time, with an index of size $O(k|V|)$ calculated in $O(k|E| \log |V|)$ time.*

Proof. Assuming we have a minimum path cover already calculated, create the reachability table by going through the nodes in topological order. Topological sorting takes $O(|V| + |E|)$ time and going through the nodes $O(k|E|)$ time, because we check each arc once. All these steps are faster than computing the minimum path cover in $O(k|E| \log |V|)$ time, which is the total time complexity for computing the reachability table. □

3.6 Running time analysis of the decomposition

In the previous sections we have explained all the details of the decomposition process components. The decomposition pipeline is illustrated in Figure 9 We have already analyzed the running times of the components individually and we will now compose the total running time of the individual running times.

Theorem 7. *Decomposing a directed acyclic graph along its maximum antichains with the method described above takes $O(k|E| \log |V|)$ time.*

Proof. Because the decomposition steps are executed in order, to find the total time complexity, we just have to find the component with largest running

time and that will be the total running time of the whole decomposition pipeline. First step, minimum path cover approximation, has running time of $O(k|E| \log |V|)$. Second step, diminishing the approximation to a minimum path cover has the same running time. Reachability table is calculated in $O(k|E|)$ time and the decomposition is found with ants in time $O(k|V|)$. The largest running time among these is $O(k|E| \log |V|)$, which is then the final time complexity.

□

4 Experiments

This section contains the results obtained from solving minimum-cost minimum path cover on various test graphs. Experiments are run always so that the problem is solved two times on each graph; once by decomposing the graph first and once by just running the solver on the original graph. This way we can measure the benefit that decomposition gives. The running times of decomposition and running the solver is summed so that the time is always the total time from the start of the decomposition until the solver finishes.

4.1 MC-MPC solver

For solving the minimum-cost minimum path cover in both decomposed and non-decomposed graphs, I have used *Network Simplex* [13] algorithm implementation from LEMON library [1][4]. I chose to use this algorithm because it was, according to my experience with test graphs, the fastest solver for this problem.

4.2 Randomly generated k-path graphs

A *k-path graph* is a model of randomly generated graphs, which resembles the graphs of our application area. K-path-graphs have small width, that is, they are guaranteed to have minimum path covers of at most size k. We will use two different k-path-graph models. The first one, which we call *forward arcs*, has minimum path cover of exactly k paths. The second model, *random arcs*, has minimum path cover of at most k paths, but it can be smaller. The

difference between these two models shows us the properties of graphs, which make the decomposition run faster and be more useful for us.

Both k-path-graph models consist of k paths of n vertices. In the context of k-path-graphs, n is the number of nodes on one path, not the total number of nodes in the graph. Nodes are indexed so that the first number is for the path the node belongs to and the second index is the node's index inside the path. The first path of a k-path-graph could be $(v_{1,1}, v_{1,2}, v_{1,3}, ..., v_{1,n})$. There is naturally an arc between every two consecutive nodes.

Until this point the two k-path-graph models are exactly same. Now we add m arcs between random nodes. In all our experiments, the value for m is equal to n. In *forward arcs*, we choose the random nodes $v_{k,n}$ and $v_{k',n'}$ so that $n < n'$. This guarantees the acyclicity, because any path in the graph will consist of nodes of increasing path indices. In *random arcs*, we don't require arcs to go from smaller path indices to bigger. Instead, we choose two nodes randomly and add the arc if there is no path from the target node to the source. As mentioned above, *random arcs* can have path cover of less than k as nothing prevents the model from generating a graph where all the nodes are on same path. That would be very unlikely, though.

The k-path-graph generator used in the measurements can be found in the repository. It creates the graphs as described above, with an additional shuffling step added to the end. After generating the graph, the generator creates the final graph from scratch, adding the nodes and arcs in random order. Shuffling does not alter the structure of the graph; it just changes the labels and the order in which nodes and arcs are added to the graph. This step is done to make sure that the decomposer or the solving algorithm cannot accidentally benefit from the order the nodes are added. Before shuffling, the nodes and arcs are added in very ordered manner, one path at time. This could lead to a situation where greedy algorithms just happen to end up in correct solutions very easily.

4.3 Results for *forward arcs*

Figure 13 show us results obtained by comparing the running times of solving with and without the decomposition. As can be seen from the results, in all cases the decomposition takes much more time than solving without it. Even when the graphs get more dense (when $k = \frac{n}{10}$), the network simplex algorithm can solve the instances with ease.

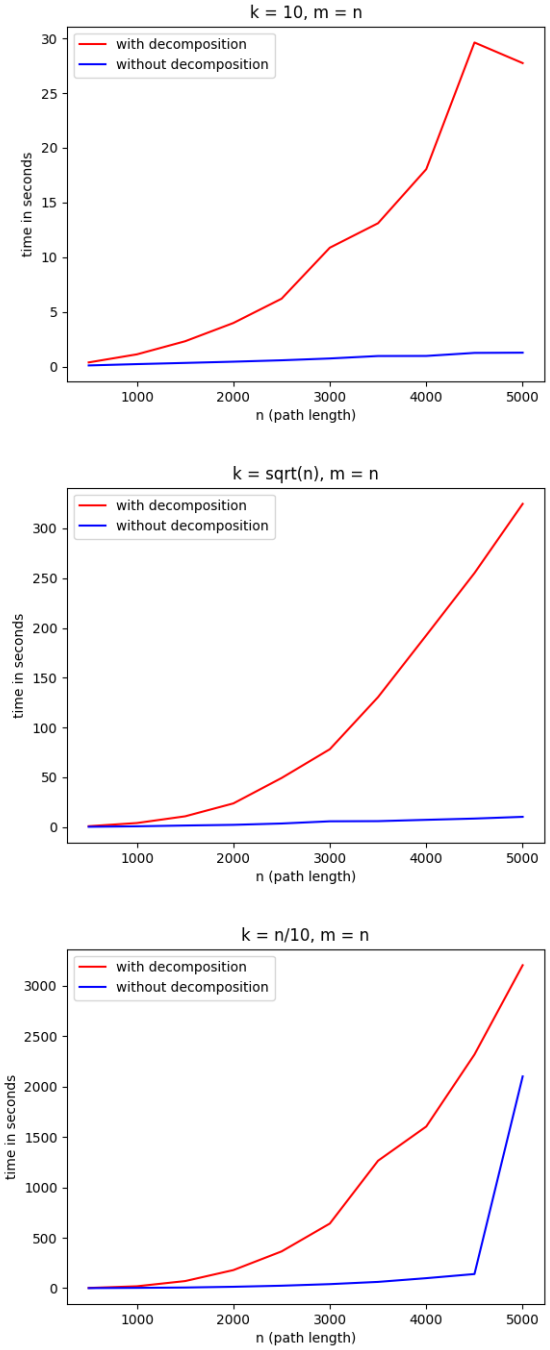

Figure 13: Performance testing on *forward-arcs* graphs with three difference values of k.

The problem the decomposer has with this model is that it cuts the graph in too many pieces. The decomposer finds always almost all of the possible decompositions, that is, n decompositions. Solving the problem in such small parts is very fast, but the high running time comes from the overhead in creating the decomposition parts where to solve the problem. In the example implementation we create a new graph object for every decomposition. When the number of decompositions is as high as it is, this process just takes too much time.

This problem could be solved by developing a method to run the solver without creating new graphs. If the decomposition could be simulated in the original graph without creating new objects, it could substantially speed up the decomposer.

4.4 Results for *random arcs*

Figure 14 show the results for solving graphs generated with *random arcs* model. Even though the decomposition does not seem to give a clear benefit in all cases, it performs significantly better than in *forward-arcs* graphs. As discussed previously, the decomposer finds too many decomposed parts in *forward arcs*. In *random arcs* the number of decompositions found is much smaller. Due to the possibility of arcs going backwards (from bigger path indices to smaller) there is usually much smaller number of maximal anti-chains in *random arcs* graphs.

4.5 Random graphs

Random directed acyclic graphs were originally not a target for this decomposition, but it is interesting to see how it performs on them. We will use a directed version of classic Erdős–Rényi model [6], where every pair of nodes in the graph has an edge between them with probability p. To make random graphs created with this model directed, we index every node, and add an arc from v_i to v_j only if $i < j$. All the arcs have starting node of smaller index and endpoint of larger index. Thus also all paths consist of nodes of increasing indices, and no paths can be cyclic.

We have results for two different values of p, 0.5 and 0.9. Those two values are chosen to represent sparse and dense random directed acyclic graphs. These results can be found in Figure 15. Decomposition seems

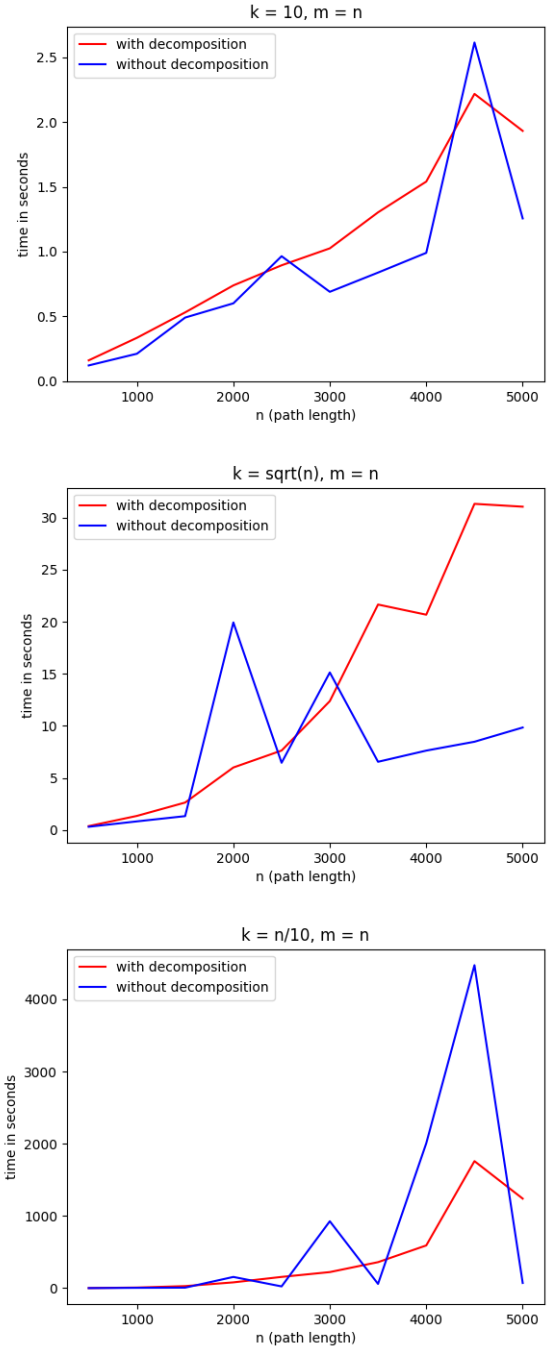

Figure 14: Performance testing for *random-arcs* graphs with three different values of k.

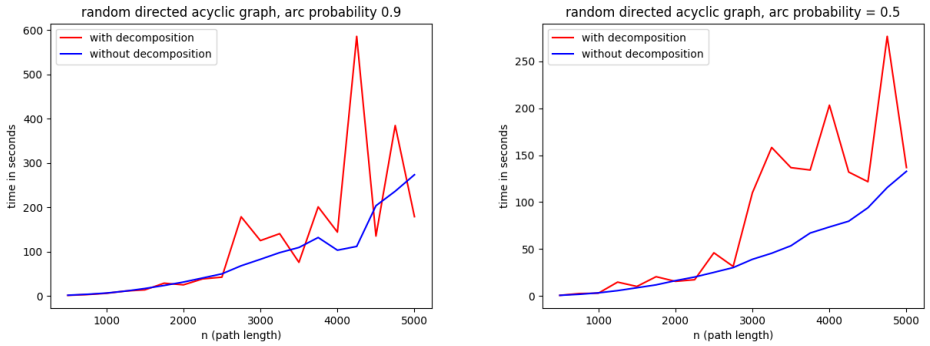

Figure 15: Comparing the running times of solvers with and without the decomposition on random dense (left) and sparse (right) graphs.

to work better than in *forward-arcs* model, but is still worse than without decomposition. There is not much difference between the performance in sparse and dense graphs.

5 Conclusion and improvement ideas

We introduced a method for decomposing directed acyclic graphs in $O(k|E|\log|V|)$ time. We also introduced a new algorithm for finding a minimum path cover in$O(k|E|\log|V|)$ time and using this algorithm we can answer reachability queries on directed acyclic graphs in $O(1)$ time, after constructing an index of $O(k|V|)$ size in $O(k|E|\log|V|)$ time. To our knowledge, the algorithm for minimum path cover is new and not presented in the literature previously. The reachability query scheme is already introduced in [2], but using the new minimum path cover algorithm we get a new space/time tradeoff. Both methods are simple, and based on fundamentals of graph theory, such as Ford-Fulkerson method for maximum flows and the classic greedy algorithm for set cover. Also, both these two methods are parametrized by the width, or the size of the minimum path cover of the directed acyclic graph. We believe that this is an important factor to our research, because graphs in gene sequencing domain tend to have small width.

We conducted experiments on randomly generated graphs and the results were shown in Section 4. As the experiments with randomly generated graphs show us, the decomposition does not give any speedup in most cases. In

forward arcs the decomposition takes much longer time than solving with library algorithms. In *random arcs* the running times are much closer to each other. Due to variance between running times on specific graph instances, it is hard to say if the decomposition would really be a beneficial pre-processing step when solving the minimum-cost minimum path cover problem. However, we can say that on some instances, decomposition speeds up the solver. The implementation of the decomposer is by no means stable or perfectly optimized. There are still bugs in the code and sometimes decomposer's answers are not correct.

There are a lot of things that could be improved in the decomposer implementation to make it run faster. One of them is to try to avoid creating new objects of the decomposed parts. According to my brief timing of the steps performed by the decomposer, creating and copying new objects seems to take a lot of time. This could be avoided by simulating the decomposition in the original graph, instead of actually creating the decomposed parts as new objects. This could also even out the difference in running times between *forward arcs* and *random arcs*, because the slowness in *forward arcs* comes from the large number of parts in the decomposition.

However, I believe that with a better implementation and further development of the ideas, this decomposition approach could be used to create more efficient solvers to minimum path cover, or even to some other problems on directed acyclic graphs. The algorithms introduced in this thesis are simple and have good running times on directed acyclic graphs of low width.

References

[1] `http://lemon.cs.elte.hu/trac/lemon`.

[2] Chen, Y. and Chen, Y.: *An efficient algorithm for answering graph reachability queries.* In *IEEE 24th International Conference on Data Engineering*, pages 893–902, 2008.

[3] Chen, Y. and Chen, Y.: *On the graph decomposition.* In *IEEE Fourth International Conference on Big Data and Cloud Computing*, pages 777–784, 2014.

[4] Dezső, B., Jüttner, A., and Kovács, P.: *Lemon – an open source c++ graph template library.* In *Electronic Notes in Theoretical Computer Science*, pages 23–34, 2011.

[5] Dilworth, R. P.: *A decomposition theorem for partially ordered sets.* In *The Annals of Mathematics*, page 51, 1950.

[6] Erdős, P. and Rényi, A.: *On random graphs.* In *Publicationes Mathematicae*, page 290–297, 1959.

[7] Ford, L. R. and Fulkerson, D. R.: *Maximum flow in a flow network.* In *Canadian Journal of Mathematics*, page 399–404, 1956.

[8] Fulkerson, D. R.: *Note on dilworth's decomposition theorem on partially ordered sets.* In *Proceedings of the American Mathematical Society*, 1956.

[9] Galvin, F.: *A proof of dilworth's chain decomposition theorem.* In *The American Mathematical Monthly, Vol. 101, No 4*, pages 352–353, 1994.

[10] Hopcroft, J. E. and Karp, R. M.: *An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs.* In *SIAM J. Comput.*, page 225–231, 1973.

[11] Kahn, A. B.: *Topological sorting of large networks.* In *Communications of the ACM*, pages 558–562, 1962.

[12] Mäkinen, V., Belazzougui, D., Cunial, F., and Tomescu, A. I.: *Genome-Scale Algorithm Design.* Cambridge University Press, 2005.

[13] Orlin, J. B.: *A polynomial time primal network simplex algorithm for minimum cost flows.* In *Mathematical Programming*, page 109, 1997.

[14] Orlin, J. B.: *Max flows in o(nm) time, or better.* In *Proceedings of the 45th Annual ACM Symposium on the Theory of Computing*, page 765–774, 2013.

[15] Pijls, W. and Potharst, R.: *Another note on dilworth's decomposition theorem.* In *Journal of Discrete Mathematics*, pages 701–702, 2013.

[16] Su, J., Zhu, Q., Wei, H., and Yu, J. X.: *Reachability querying: Can it be even faster?* In *IEEE Transactions on Knowledge and Data Engineering*, page 683–697, 2017.

[17] Vazirani, V. V.: *Approximation Algorithms.* Springer-Verlag, 2001.